Universidad de Castilla-La Mancha

Escuela Superior de Informática

Memoria del laboratorio de Diseño de Algoritmos

Autores: *Alejandro del Hoyo Abad, Sergio Pozuelo Martín Consuegra y Alberto Barraís Bellerín*
Grupo: G02
Asignatura: Diseño de algoritmos
Fecha: 14-04-2024

Índice

1. Introducción

Durante este laboratorio, hemos ido desarrollando un proyecto que ha consistido en desarrollar un juego de mesa poco conocido llamado *Hnefatafl* (el ajedrez de los vikingos). En este sentido, el propósito definitivo del laboratorio ha sido el de implementar de forma práctica los distintos algoritmos que se han ido viendo en teoría. De esta forma, hemos sido capaces de poner en práctica nuestros conocimientos en programación de forma satisfactoria en programación. Por lo tanto, durante esta memoria, se explicarán las distintas partes que se han tenido en cuenta para poder llevar a cabo este proyecto de forma satisfactoria, comenzado por una breve explicación de en que consiste el juego, su representación, los algoritmos implementados y las mejores que se han llevado a cabo para poder realizar de forma satisfactoria todas estas fases.

2. Juego del Hnefatafl

El juego del Hnefatafl es un juego que consiste en un tablero cuadriculado, normalmente de tamaños 11x11 o 9x9, pero puede haber más versiones. En este caso, se enfrentan dos jugadores, siendo las fichas negras que atacan cuya tarea es la de capturar al rey y las blancas que defienden al rey. En este sentido, las piezas atacantes son las primeras en moverse. El movimiento de todas las fichas es igual al movimiento de las torres en el ajedrez, en donde no se puede saltar sobre otras piezas. Asimismo, las fichas no podrán posicionarse en las esquinas del tablero, véase las casillas de escape, ni en la casilla central, en la cual solo puede estar el rey.

La captura de una ficha normal se realiza al mover una pieza de un jugador a un espacio adyacente ocupado por una pieza del oponente, siempre que este último esté flanqueado por dos piezas del jugador activo, simulando así una emboscada táctica. Asimismo, las casillas de escape se podrán usar para realizar las respectivas capturas, al igual que la del trono (si no está el rey funciona como las de escape). Estas capturas suceden tanto de forma horizontal como en vertical.

Por último, en cuanto a las condiciones de victoria, será necesario distinguir el tipo de ficha con el que se está tratando. Por un lado, para las fichas blancas debe ocurrir que el rey llegue a una de las casillas de las esquinas o que solo quede 1 o ninguna ficha negra (en estos casos ya no se podrá realizar la respectiva captura). Por otro lado, las fichas negras ganarán cuando sean capaces de rodear por todos los lados a las blancas (se podrá utilizar la casilla del trono). Por último, se ha tenido en cuenta una condición de empate que tiene lugar cuando las posiciones en las que se encuentra el tablero en un momento dado son exactamente iguales a 8 jugadas anteriores.

2.1. Representación del Juego

En cuanto a la representación del juego, destacamos los siguientes apartados:

Estructura de Representación

Para la representación del tablero en el juego Hnefatalf, se ha elegido utilizar una lista unidimensional en lugar de una matriz bidimensional. Esta decisión se basa en la eficiencia de memoria y la velocidad de acceso a las posiciones específicas del tablero durante el juego. En un archivo JSON denominado INITSTATES.json, se definen diferentes configuraciones de juego, incluyendo la disposición inicial de las piezas para cada variante del juego.

Formato JSON para configuraciones del juego

La estructura JSON para almacenar la configuración de los juegos es la siguiente:

```
{
   "name": "Brandubh",
   "size": 7,
   "escape": [0, 6, 42, 48],
   "center": 24,
   "init_state": {
      "ID": "0",
      "white": [17, 23, 25, 31],
      "black": [3, 10, 21, 22, 26, 27, 38, 45],
      "king": 24,
      "gamer": 0
   }
}
```

```
00 01 02 03 04 05 06
07 08 09 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31 32 33 34
35 36 37 38 39 40 41
42 43 44 45 46 47 48
```

Tablero 7x7

Detalles de la configuración

- **name**: Identifica la variante del juego, permitiendo múltiples configuraciones y reglas.

- **size**: Define el tamaño del tablero, que es un cuadrado, resultando en un total de 49 casillas para un tablero de 7x7.

- **escape**: Identifica las casillas de escape que son críticas para la victoria del rey.

- **center**: Especifica la posición central del tablero, que puede tener significados especiales en diferentes variantes del juego.

- **init_state**: Define el estado inicial del juego, incluyendo la disposición de las piezas blancas, negras y el rey, así como el jugador inicial.

Ventajas de la representación unidimensional

- **Eficiencia de acceso**. Acceder a cualquier casilla del tablero es directo mediante un índice, en lugar de requerir dos índices como en una matriz bidimensional.

- **Simplicidad de manejo**. La manipulación de las posiciones es más sencilla, ya que operaciones como mover piezas o verificar el estado del juego se pueden realizar con menos cálculos y comparaciones.

- **Optimización de memoria**. Al almacenar el tablero en una lista lineal, se optimiza el uso de la memoria, lo cual es especialmente beneficioso en entornos con recursos limitados.

Implementación en código

- **Clase Board**. Gestiona la configuración del tablero, como tamaño y casillas especiales. Utiliza métodos como *initialize* para configurar estas propiedades basadas en los datos cargados desde INITSTATES.json.

- **Clase State**. Representa un estado del juego, almacenando las posiciones de todas las piezas y el turno del jugador. También gestiona la lógica para cambiar de turno, generar sucesores de estado y realizar movimientos válidos.

2.2. State (Representación de estados)

State gestiona la lógica y representación individual de cada estado de juego, incluyendo la representación gráfica y comprobación de efectos de los movimientos

Propiedades y Constructor

- **Propiedades**. Contiene la posición de todas las fichas en el tablero, así como el turno del jugador tipo de juego, movimientos pasados y movimientos posibles.

- **Constructor**. Inicializa las posiciones, tipo de juego y turno a los valores proporcionados. En caso de que se incluya un String de estado en formato JSON, el constructor reemplazará todos los valores establecidos por los datos extraídos de este. Una vez completado, si el estado tuviese una id = 0, simbolizando que no tiene id asignada, se crearía una nueva a partir de un método

Representación de datos

- **Fichas blancas**. Posiciones donde se encuentra una ficha blanca. Se guardan como una lista de enteros representando el cuadrado donde se encuentran.

- **Rey**. Posición en la que se encuentra el rey. Se guarda como un entero representando el cuadrado en el que se encuentra.

- **Fichas negras**. Posiciones donde se encuentra una ficha negra. Se guardan como una lista de enteros representando el cuadrado donde se encuentran.

- **Jugador**. Indica el jugador que debe de mover ficha en un determinado turno. Se representa como un entero 0 o 1 para indicar que es el turno de las fichas negras o blancas respectivamente.

- **Movimientos posibles**. Movimientos legales que se pueden realizar. Se representa como una lista de tuplas (posición actual, lista de posiciones válidas) donde se guarda el cuadrado donde hay una ficha y una lista con las posiciones a la que se podría mover esa ficha.

- **Movimientos pasados**. Últimos 8 movimientos que se han realizado. Se utiliza para calcular los empates. Se representa como una lista de Strings que guardan las ID de los últimos 8 movimientos.

Funcionalidades principales

- **Buscar movimientos posibles**: mov_valid_list() genera una lista de todos los movimientos posibles en el tablero.

- **Generar sucesores**: generate_successors() crea una copia profunda de sí mismo por cada movimiento posible y le hace los cambios correspondientes según el movimiento en la lista de movimientos posibles.

- **Ejecutar movimientos**: make_movement() permite mover la posición de una pieza dada su posición actual y su posición final, comprobando las capturas en el proceso. handle_click() se encarga de procesar los movimientos realizados por jugadores humanos a traves de la interfaz.

- **Representación gráfica**: draw_board(), draw_special_squares() y draw_pieces() se encargan de pasar la información de un estado a una interfaz gráfica con la que pueda interactuar el usuario. Estas funciones crean el tablero, las posiciones especiales y las fichas respectivamente.

2.3. Clase board (Representación de tablero)

Representa internamente las características y posiciones especiales del tablero de juego. Los datos de esta clase se guardan como variables de clase en lugar de variables objeto para que se compartan en cualquier instancia, permitiendo acceder a ellos desde cualquier objeto Board para facilitar su acceso.

Representación de datos

- Tamaño del tablero. Se representa el tamaño de uno de los lados como un número entero. Cuando se necesite este dato, se multiplicará por sí mismo para obtener el número total de cuadrados en el tablero.

- Posiciones de escape. Posiciones a las que se tiene que mover el rey para contar como victoria de las fichas blancas. Se representa como una lista de enteros indicando el cuadrado concreto en el que están. Suelen ser las esquinas.

- Trono. Casilla especial de la que parte el rey al iniciar la partida que no puede ocupar ninguna otra pieza. Se representa como un entero que indica el cuadrado concreto en el que está. Suele ser el centro.

Inicialización

El constructor se crea en una función distinta a la que usualmente se usaría (init) para que la asignación de valores sea manual y no se necesiten de los datos para crear una instancia. Requiere de los valores para los tres datos que almacena.

3. Game (Gestor de partida)

La clase Game gestiona la lógica de alto nivel del juego, coordinando el flujo del juego, incluyendo la inicialización del tablero y los estados, y la interacción entre los jugadores y el tablero.

Propiedades y Constructor

- Propiedades: Incluye la variante del juego (type_game), un objeto Board, el estado actual del juego (current_state) y los jugadores (black_player, white_player).

- Constructor: Carga la configuración del juego desde INITSTATES.json, inicializa el tablero y establece el estado inicial del juego.

Funcionalidades clave

- Juego Gráfico: *graphic_board()* dibuja el tablero y las piezas en la interfaz gráfica, permitiendo una interacción visual con el juego.

- Simulación del Juego: *play_game()* ejecuta la lógica del juego, alternando turnos entre los jugadores y actualizando el estado del juego.

4. Servidor (Gestor de juegos)

El componente *servidor* en la implementación del juego Hnefatalf no representa un servidor en el sentido convencional de redes, sino que más bien actúa como un gestor centralizado del juego, encargado de crear partidas, administrar jugadores y registrar los resultados.

Función principal del servidor

El servidor, implementado a través de la *clase Server*, se encarga de:

- Iniciar y gestionar las partidas del juego.

- Registrar y actualizar las estadísticas de los jugadores.

- Guardar las partidas para posibles repeticiones o análisis futuros.

Inicialización y configuración

- Inicio del servidor: Al instanciar, el servidor registra su hora de inicio y prepara el directorio GAMES donde se guardarán los archivos de las partidas.

- Creación de Partidas: Mediante *new_game(game_name)*, se puede iniciar una nueva partida, asignando un nombre único y generando un ID de juego basado en la fecha, hora y nombre del juego.

Gestión de jugadores

- Asignación de jugadores: *join_game(gamer, player_name)* permite asignar jugadores a la partida. Los jugadores pueden ser humanos o varios tipos de CPU que emplean diferentes estrategias de juego, como algoritmos aleatorios, Minimax, Monte Carlo o Q-Learning.

- Excepciones de jugadores: Se manejan casos donde ya existe un jugador asignado a un color específico o si falta alguno para iniciar la partida.

Ejecución y resultados de la partida

- Inicio de la partida: *start_game()* verifica que ambos jugadores estén asignados antes de comenzar la partida.

- Registro de resultados: *result(winner)* registra el resultado de la partida, actualiza las estadísticas de los jugadores y guarda los resultados en un archivo JSON.

Reproducción y Análisis de Partidas

- Repetición de partidas: *replay(game_data)* permite cargar y reproducir una partida desde un archivo de datos, útil para análisis o revisión.

- Reset del servidor: *reset_server()* limpia las configuraciones y prepara el servidor para una nueva partida.

Estructura de datos y excepciones

- Almacenamiento de Resultados: *parse_results()* convierte los resultados de la partida en una cadena JSON, facilitando su almacenamiento y posterior análisis.

- Excepciones personalizadas: se definen excepciones como Player_Already_Joined y Missing_Players para manejar situaciones específicas relacionadas con la asignación de jugadores.

5. Agentes

En este apartado se explicarán todos los agentes que se han implementado durante el desarrollo del proyecto.

5.1. Clase padre “player”

La clase *Player* actúa como la clase base para todos los tipos de agentes que participan en el juego. Implementada como una clase abstracta, define una interfaz común y establece un marco que todas las clases derivadas deben seguir, asegurando que todos los jugadores se comporten de manera coherente bajo un mismo conjunto de reglas y estructuras.

Asimismo, en cuanto a la estructura de esta clase, podemos destacar los siguientes aspectos:

Propiedades básicas:

- *state*: Mantiene una referencia al estado actual del juego, que contiene el tablero y la posición de todas las piezas.

- *directory*: Define el directorio donde se almacenan los archivos de estadísticas del jugador.

Métodos fundamentales:

- *info()*: Método abstracto diseñado para proporcionar información del jugador. Aunque está vacío en la clase base, permite que las subclases personalicen la información que devuelven. Es fundamental para el correcto aprendizaje del Q-learning.

- *load_stats()* y *save_stats()*: Métodos para cargar y guardar las estadísticas del jugador desde y hacia archivos JSON. Estos métodos permiten el seguimiento del desempeño del jugador a lo largo del tiempo.

- *parse_stats():* Convierte las estadísticas del jugador en una cadena JSON, facilitando su almacenamiento y recuperación.

- *md5_id()*: Genera un identificador único MD5 para el jugador a partir de su nombre, útil para manejar identificaciones únicas en entornos sin conexión.

Movimientos:

- *make_movement()*: Método que actualiza el estado del juego ejecutando un movimiento. Depende del *método _next_movement()*, que es abstracto y debe ser definido en cada subclase para especificar cómo el jugador decide su próximo movimiento.

5.2. Humano

El Agente Humano no es un agente, es la implementación del player para los usuarios. Permite a los jugadores interactuar con el juego a través de una interfaz gráfica, seleccionando movimientos mediante clics de ratón. Este agente es crucial para proporcionar una experiencia de juego interactiva y tangible.

Implementación

Utiliza el módulo *pygame* para manejar eventos de entrada del usuario, lo que permite que los jugadores humanos hagan movimientos manualmente. Este método implica esperar y procesar eventos de clics del ratón para determinar el movimiento deseado en el tablero.

Características y funciones

- **Interacción usuario**: Permite la entrada de movimientos mediante interacción directa del usuario usando el mouse.

- **Gestión de eventos**: Utiliza eventos de pygame para detectar y procesar clics del mouse para realizar movimientos.

5.3. Aleatorio

El Agente Aleatorio representa la forma más básica de agente IA, proporcionando un nivel de oponente básico y predecible. Sirve como un oponente automático que elige movimientos al azar de la lista de movimientos válidos, proporcionando un nivel de desafío básico y la capacidad de simular partidas para pruebas.

Implementación

Genera movimientos seleccionando aleatoriamente entre las opciones de movimientos válidos disponibles en el estado actual del juego, lo cual se realiza usando la función *random.choice()* sobre la lista de posibles movimientos.

Características y funciones

- **Selección Aleatoria**: Elige movimientos de forma aleatoria de la lista de movimientos válidos proporcionada por el estado del juego.

5.4. Minimax

El Minimax es un algoritmo clásico de decisión para juegos de dos jugadores basado en la simulación de movimientos alternos hasta una profundidad especificada, evaluando el estado final del juego para retroceder y seleccionar el mejor movimiento posible.

Implementación

El agente Minimax profundiza en el árbol de juego hasta un límite establecido que es el *max_depth* (en nuestro caso tiene de forma predeterminada profundidad 50 aunque puede aumentarse), utilizando la poda alfa-beta para optimizar el proceso de búsqueda. Se evalúan los estados del juego con una función heurística, que favorece las posiciones ventajosas. Del mismo modo, Las funciones max_value y min_value se encargan de calcular los valores para el jugador que busca maximizar y minimizar, respectivamente, utilizando una estrategia de búsqueda en árbol recursiva. Sin embargo, para mejorar la eficiencia, se aplica poda alfa-beta en estas funciones, lo que significa que se evitan explorar ramas de árbol que sabemos que no afectarán el resultado final. Finalmente, la función search_alpha_beta selecciona el mejor movimiento posible para el jugador actual explorando los sucesores del estado actual y aplicando el algoritmo Minimax con poda alfa-beta. Este enfoque permite una búsqueda más eficiente de movimientos en juegos complejos al evitar la exploración innecesaria de ramas del árbol de juego.

5.5. Montecarlo

El *Montecarlo Tree Search (MCTS)* es un método avanzado que usa simulaciones aleatorias para tomar decisiones basadas en el cálculo estadístico de las recompensas acumuladas en nodos explorados, optimizando la selección de movimiento.

Implementación

Este método construye un árbol de búsqueda de forma incremental, simulando partidas desde el estado actual hasta estados terminales y retropropagación de los resultados para informar decisiones futuras. El proceso se repite múltiples veces para refinar la selección de movimientos. En este caso, hemos puesto de forma predeterminada 600 iteraciones pero se pueden aumentar para que sea más preciso a la hora de elegir el siguiente movimiento.

Características y Funciones

- **Selección y expansión de nodos**: Maneja la expansión de nodos en el árbol de búsqueda y selecciona el siguiente movimiento basado en el valor UCT (Upper

Confidence bounds applied to Trees). Más concretamente, el método *tree_policy()* comienza en el nodo raíz del árbol y continúa iterando hasta que se alcanza un estado terminal. En cada iteración, intenta expandir el nodo actual, generando un nuevo nodo hijo que representa un posible estado sucesor del juego. Si la expansión es exitosa y se genera un nuevo nodo hijo, este nodo hijo se devuelve como el próximo nodo a explorar. Si no es posible expandir más el nodo actual, se selecciona el mejor hijo del nodo actual utilizando el correspondiente criterio UCT. Finalmente, se devuelve el nodo que debe de expandirse.

- **Monte Carlo Tree Search (MCTS)**: Usa simulaciones aleatorias para explorar los resultados más prometedores de los movimientos posibles. Estas simulaciones son para los nodos que han sido expandidos en donde se simulará aleatoriamente una partida en donde se devolverá 1 para el caso de victoria, 0 para empate y -1 para derrota. Las victorias y las derrotas son con respecto al tipo de ficha que está utilizando el algoritmo de Montecarlo durante una partida.

- **Simulación y Retropropagación**. Retropropaga los resultados a los padres de los respectivos nodos de forma recursiva hasta que se llegue al nodo raíz.

5.6. MiniMax con optimización por elección aleatoria

La clase *PlayerMiniRandom* introduce una variante del algoritmo Minimax en donde se mezcla la evaluación estratégica con elementos de aleatoriedad. Este enfoque no solo nos permite maximizar la eficacia de los movimientos basándose en una evaluación heurística, sino que también agiliza los primeros estados donde es difícil hayar una camino mediante las elecciones aleatorias.

Implementación

- **Proceso de Decisión**: Primero evalúa todos los posibles sucesores del estado actual usando Minimax con poda alfa-beta hasta una profundidad especificada (*max_depth*). Luego, clasifica estos movimientos basados en su valor evaluado.

- **Selección de Movimientos**:

    - Si todos los movimientos tienen un valor de cero o no hay movimientos claramente ventajosos, selecciona aleatoriamente entre ellos, lo que ayuda a evitar patrones predecibles (ya que, si no se hiciera esto, cuando no encuentra un movimiento óptimo, elegiría siempre el primer movimiento posible) y potencialmente explotables.

    - Si hay movimientos con valores positivos, selecciona el movimiento con valor más alto. En que de haber varios movimientos con el mismo valor, siendo este valor el más alto, selecciona aleatoriamente entre ellos.

- **Gestión del Tiempo**: Utiliza un límite de tiempo (max_time) para asegurar que las decisiones se tomen rápidamente, lo cual es crucial para un juego fluido y para mantener la carga computacional a un nivel manejable.

- **Evaluación de Estado**: La función de evaluación considera tanto la situación inmediata del tablero como posibles desarrollos futuros, ajustando las evaluaciones basadas en la profundidad del análisis y el jugador actual.

Características y funciones

- **Algoritmo Minimax**: Implementa el algoritmo Minimax para tomar decisiones estratégicas, mirando varios movimientos hacia adelante.

- **Poda Alfa-Beta**: Utiliza la técnica de poda alfa-beta para reducir el número de nodos evaluados en el árbol de juego.

- **Evaluación de estado**: Define una función heurística para evaluar los estados del juego desde la perspectiva de maximizar las ventajas.

5.7. Q-Learning (Aprendizaje)

Q-Learning es una técnica de aprendizaje por refuerzo que permite a los agentes aprender políticas óptimas de juego a través de la experiencia, sin un modelo del ambiente. Adapta sus estrategias basándose en recompensas obtenidas por acciones previas.

Implementación

El agente mantiene una tabla Q que mapea estados a valores de acción, actualizándola con cada acción tomada basándose en las recompensas recibidas y los valores futuros estimados. Este enfoque permite que el agente adapte sus estrategias a medida que gana experiencia en el juego.

Características y funciones

- **Tabla Q (*q_table*)**: Mantienen una tabla Q que asocia estados con valores Q para cada posible acción, ajustando estos valores a medida que se aprende más sobre el entorno del juego.

- **Política de exploración**: Alterna entre explorar nuevas acciones y explotar conocimientos adquiridos para optimizar la selección de movimientos.

Heurística en común

La heurística que compartes ambas implementaciones es a la hora de puntuar la victoria. La recompensa será más alta cuanto menos movimientos necesite para ganar a su contrincante.

Implementación de la heurística

Si el estado actual indica que la tarea está finalizada (*self.state.is_finished()*), entonces se asignan recompensas según la cantidad de iteraciones (*self.iteraciones*) que ha realizado el agente.

- Si el agente ha realizado 50 o más iteraciones, se le asigna una recompensa de 50.
- Si ha realizado entre 30 y 49 iteraciones, se le asigna una recompensa de 80.
- Si ha realizado entre 15 y 29 iteraciones, se le asigna una recompensa de 100.
- Si ha realizado entre 5 y 14 iteraciones, se le asigna una recompensa de 150.
- Si ha realizado menos de 5 iteraciones, se le asigna una recompensa de 200.

Heurística blancas

El objetivo de la heurística en las blancas es optimizar el movimiento del rey hacia posiciones de escape, premiando movimientos que acerquen al rey a los bordes del tablero, que son puntos críticos para ganar el juego.

Para llevar a cabo esto, serán necesarias utilizar las siguientes funciones:

- *policy()*. Esta función calcula una recompensa basada en la proximidad del rey a las casillas de escape. Utiliza la distancia euclidiana entre la posición actual del rey y las casillas de escape más cercanas. La recompensa se escala inversamente con la distancia, de modo que cuanto más cerca esté el rey de un punto de escape, mayor será la recompensa.

- *policyhatediagonal()*. Esta función ajusta la recompensa teniendo en cuenta la posición diagonal del rey en relación con las esquinas del tablero. Si el rey está en una

diagonal y no está en una casilla de escape o adyacente, recibe una penalización porque las diagonales son típicamente rutas menos directas para escapar. Sin embargo, si está en un borde (y más cerca de una escapatoria), se le da una recompensa más alta.

Heurística negras

Para las negras, las heurísticas están diseñadas para bloquear o capturar al rey, ya que su objetivo es prevenir que el rey escape. En este sentido, al ser más difícil las victorias de las negras, al contrario que con las blancas, puntuamos los empates positivamente.

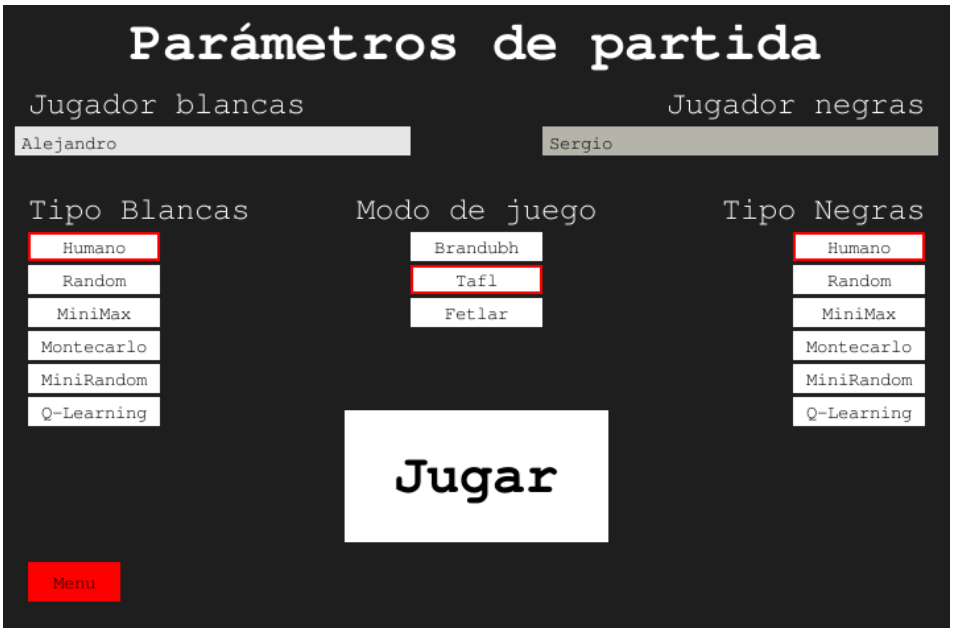
6. Manual de usuario

En esta parte, se explicará la manera sencilla los pasos necesarios para poder crear un juego sin problemas, ver las estadísticas de un jugador y ser capaz de visualizar la repetición de una partida que ya ha sido jugada.

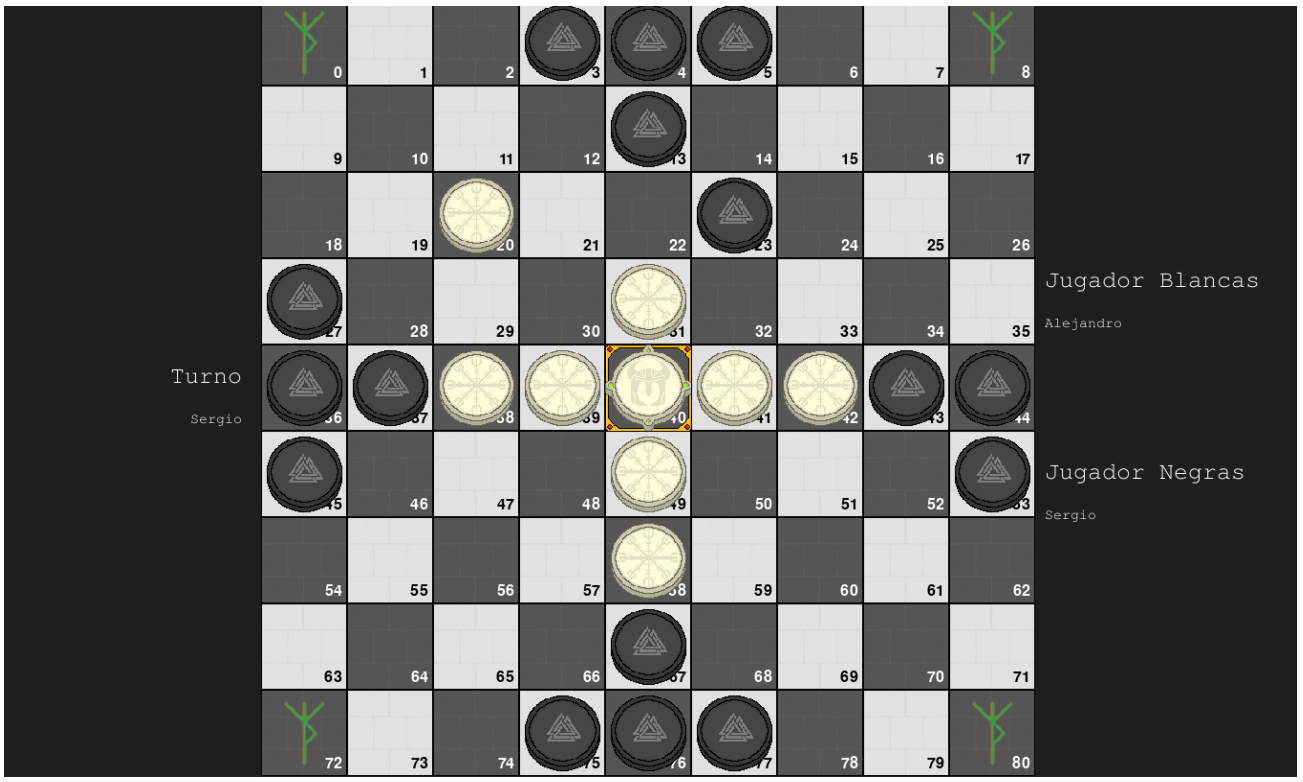
En primer lugar, será necesario posicionarse en el directorio *lab-2,* que es donde se encuentra todas las carpetas que hemos utilizado para llevar a cabo nuestro proyecto. En este sentido, para que nos aparezca la ventana del menú principal, deberemos de ejecutar el siguiente comando: *python3 src/main.py.* En este sentido, cómo se ha mencionado previamente, nuestro programa usa de las librerías de pygame y tkinter, por lo que tendrán que estar previamente instaladas para poder ejecutar nuestro proyecto. Por lo tanto, teniendo esto en cuenta, se nos mostrará la siguiente interfaz:

Menú principal

Del mismo modo, para poder crear una partida, se deberá de marcar la opción de jugar, la cual nos conducirá a una ventana como esta:

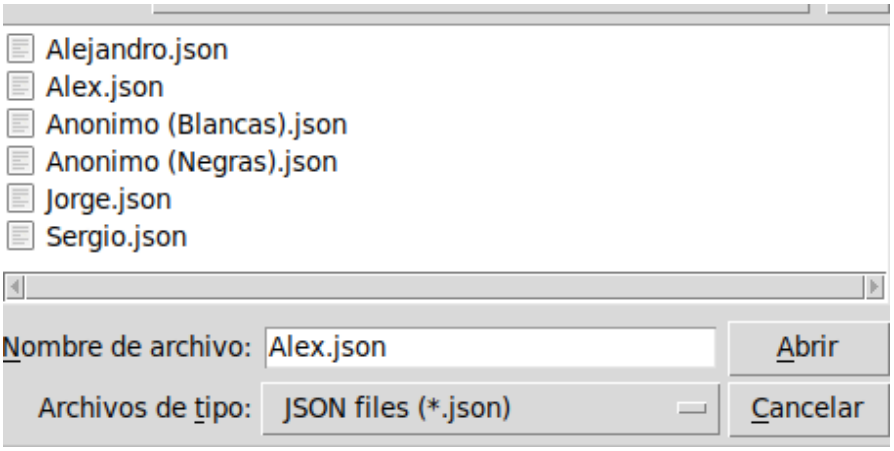


En este caso, se podrá elegir el modo de juego deseado (tamaño del tablero) junto con el tipo de agente con el que se desea jugar. En este sentido, una vez pulsado el botón de jugar, nos aparecerá el correspondiente tablero en donde se podrán enfrentar lo dos agentes.

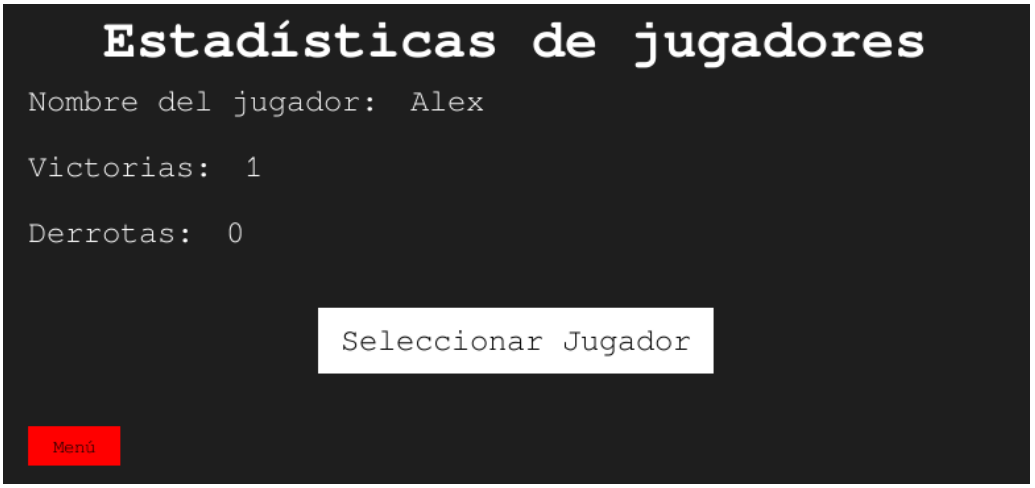


Simulación de partida

Por otro lado, para ver las estadísticas de un jugador, deberemos de dirigirnos al botón de estadísticas. En este caso, aparecerá la siguiente pestaña en donde deberemos de elegir uno de los .json que se encuentran en la carpeta PLAYERS.



Una vez seleccionado el correspondiente *json*, seremos capaces de ver las estadísticas que tiene ese jugador concreto de la siguiente forma:



Por último, la sección de repeticiones es muy similar a la de estadísticas. En este caso, se deberá de seleccionar un *json*, como se hizo para la parte de repeticiones, pero en este caso se elegirá desde la carpeta GAMES para un juego que queramos analizar.

Una vez seleccionado el correspondiente archivo que contiene el juego, podremos ver la correspondiente secuencia de movimientos que se han llevado a cabo de forma visual.

7. Conclusiones

En esta sección, expondremos nuestros pensamientos y conclusiones sobre este proyecto.

7.1. Alejandro

Personalmente, durante la realización de este laboratorio he sido capaz de aprender muchos conocimientos interesantes, principalmente sobre algoritmos. En este sentido, tengo que admitir que uno de los puntos más interesantes a la hora de llevar a cabo la implementación del proyecto fue la de organizar el código, puesto que, al ser un proyecto medianamente grande comparado con otros que hemos hecho en la carrera, ha sido necesario un mayor nivel de coordinación y cooperación. Asimismo, la comunicación y el reparto de tareas entre todos los integrantes del grupo has sido completamente fundamental, puesto que ha habido ocasiones en donde se ha modificado una parte del código que afectaba a otra, lo que ha supuesto en diversas ocasiones algunos conflictos. No obstante, ha sido satisfactoria como finalmente todos los errores y correspondientes bugs se han ido solventando, además de como se ha ido mejorando el juego y los respectivos algoritmos. No obstante, la principal pega del juego en cuestión deriva en lo que ya se ha ido comentando en clase, es decir, el juego es asimétrico, puesto que las condiciones de victoria de un tipo de fichas u otras son distintas.

7.2. Sergio

En mi opinión, el laboratorio ha sido interesante y me ha ayudado a entender mejor los algoritmos trabajados. Sin embargo, cosas como la indecisión general acerca de las reglas aun a mitad del proyecto ha causado retrasos por la necesidad de reescribir constantemente partes del código. Por otra parte, opino que las reglas elegidas al final han entorpecido el desarrollo del algoritmo Q-learning al dar una ventaja decisiva a uno de los jugadores. En general, creo que se podría mejorar con respecto a los próximos años con la elección de un juego más adecuado, simétrico y con un número de estados más manejable o la implementación de una heurística adecuada. Pese a todo, es una manera más dinámica y entretenida de estudiar los algoritmos pertinentes que permite entenderlos en más profundidad. El trabajo en equipo, salvo algún percance con pequeños retrasos y desafíos a la hora de unir las ramas de trabajo en el repositorio, ha funcionado perfectamente.

7.3. Alberto

Para mí el laboratorio ha estado bien, pero ha tenido algunos fallos. La falta de simetría dificulta que los bots muestren su potencial sin el uso de heurísticas. Además, el gran tamaño de posibilidades entorpece el aprendizaje del algoritmo Q-learning. Aunque como ya he dicho antes, si se aplican heurísticas esto podría mejorar bastante. En cuanto al trabajo en grupo, salvo algún inconveniente con retrasos u errores en implementaciones, se ha llevado a cabo de forma satisfactoria. Lo único que no he visto adecuado es la elección del juego, ya que, si hubiera sido uno simétrico o con menos estados, se podría haber conseguido un resultado final superior.